利用 Sun Studio C 编译器实现 C 代码并行化

阎　蕊　曾洁玫　王艳萍
（北京航天飞行控制中心）

摘　要　Sun C 编译器提供了在多处理器机器上优化代码的功能，被优化的代码可以使用系统上的多个处理器并行执行，从而大幅提高性能。通过研究编译器的并行化功能，告诉应用软件分析并行化的约束、并行化有依赖关系的循环，改进循环并行化以提高程序性能，以及进行显式并行化，并提出并行化建议，从而为应用软件提高效率提供了一个途径。

关键词　Sun C　编译器　并行化

1 概 述

C 编译器为它确定可以安全进行并行化的循环生成并行代码。这些循环的迭代彼此独立，迭代以什么顺序执行或者是否并行执行并不重要。许多向量循环都属于此类。由于 C 中使用别名的方式，难以确定并行化的安全。为帮助编译器确认，Sun C 提供了 pragma 和附加指针限定，以提供编程人员知道但编译器无法确定的别名信息。可以使用% cc -fast -xO4 -xautopar example.c -o example 启用和控制并行化 C。

2 与并行化相关的环境变量

与并行化 C 相关的环境变量有四种：PARALLEL，SUNW_MP_THR_IDLE，SUNW_MP_WARN，STACKSIZE。

PARALLEL 环境变量指定可供程序使用的处理器数。如果目标机器具有多个处理器，线程可以映射到独立的处理器。

一旦为当前程序的起始线程创建辅助线程，辅助线程将会参与执行程序的并行部分（并行循环、并行区域等），并在程序的串行部分运行时保持自旋等待状态。在程序终止之前，这些绑定线程不会休眠或停止。并行化程序在专用系统上运行时，使这些线程保持自旋等待状态通常可达到最佳性能。然而，保持自旋等待的线程会占用系统资源。使用 SUNW_MP_THR_IDLE 环境变量控制每个线程在完成其并行作业后的状态。默认值为 spin，表示线程在完成并行任务之后应自旋（或忙等待），直到新的并行任务到来时为止。另一个选项 sleep[n sln ms] 在自旋等待 n 个单位之后使线程进入休眠状态。等待单位可以是秒（s，默认单位）或毫秒（ms），其中 1s 表示 1 秒，10ms 表示 10 毫秒。不带参数的 sleep 在线程完成并行任务后使线程立即进入休眠状态。sleep、sleep0、sleep0s 以及 sleep0ms 均等价。如果新作业在未达到 n 个单位之前到达，线程将停止自旋等待并开始执行新作业。

将 SUNW_MP_WARN 环境变量设置为 TRUE 时，可以从 OpenMP 或其他并行化运行时系统打印警告消息。如果通过使用 sunw_mp_register_warn() 登记某个函数以处理警告消息，则即使将 SUNW_MP_WARN 设置为 TRUE，它也不会打印警告消息。如果未登记函数，并且将 SUNW_MP_WARN 设置为 TRUE，则 SUNW_MP_WARN 将警告消息打印至 stderr。

正在执行的程序会为主线程保留一个主内存栈，同时为每个从属线程保留不同的栈。栈是临时内存地址空间，用来存储子程序调用中的参数和自动变量。主栈的默认大小约为 8 兆字节。可以使用 limit 命令显示并设置当前主栈大小。多线程程序

的每个从属线程均具有其自身的线程栈。该栈与主线程的主栈相似,但对该线程来说是唯一的。在线程栈中进行分配线程的专用数组和变量(对于线程是局部的)。所有从属线程都有同样的栈大小,默认情况下,对于 32 位应用程序为 4MB,对于 64 位应用程序为 8MB。可以用环境变量 STACKSIZE 来设置该大小:对于某些已并行的代码,可能需要将线程栈大小设置为比默认值大的值。有时编译器会生成警告消息,指出需要更大的栈大小。除了通过尝试并出错之外,不可能知道应设置多大的栈大小正合适,尤其是涉及私有/局部数组时。如果栈大小太小而导致线程无法运行,程序将异常终止并出现段故障。

3 并行化的约束

C 编译器通过分析程序中的循环来确定并行执行循环的不同迭代是否安全。分析的目的是确定循环的两次迭代之间是否会相互干扰。通常,如果变量的一次迭代读取某个变量而另一次迭代正在写入该变量时,会发生干扰。例如:

```
for (i=1; i < 10; i++){sum = sum + a[i];}
```

在例子中,任意两次连续迭代 i 和 i+1 将写入和读取同一变量 sum。因此,为了并行执行这两次迭代,需要以某种形式锁定该变量。否则,允许并行执行这两次迭代不安全。然而,使用锁定会产生可能降低程序运行速度的开销。C 编译器通常并不会并行化上例中所示循环。在下面例子中,循环的两次迭代之间存在数据依赖性。

```
for (i=1; i < 10; i++){a[i] = 2 * a[i];}
```

在此情况下,循环的每次迭代均引用不同的数组元素。因此,循环的不同迭代可以按任意顺序执行。由于不同迭代的两个数据元素不可能相互干扰,因此它们可以并行执行而无需任何锁定。

编译器为确定一个循环的两次不同迭代是否引用相同变量而执行的分析称为数据依赖性分析。如果其中一个引用写入变量,数据依赖性阻止循环并行化。编译器执行的数据依赖性分析有三种结果:

·存在依赖性。在此情况下,并行执行循环不安全。

· 不存在依赖性。循环可使用任意多个进程安全地并行执行。

·无法确定依赖性。为安全起见,编译器假定存在阻止并行执行循环的依赖性,并且不会并行化循环。

下面例子中,循环的两次迭代是否写入数组 a 的同一元素取决于数组 b 是否包含重复元素。除非编译器可以确定实际情况,否则它假定存在依赖性并且不会并行化循环。

```
for (i=1; i < 10; i++){a[b[i]] = 2 * a[i];}
```

4 并行化有依赖关系的循环

循环的并行执行由 Solaris 线程完成。程序启动时,主线程创建多个从属线程。程序结束时,所有从属线程均终止。从属线程的创建只进行一次,以使开销减至最小。程序启动后,主线程开始执行程序,而从属线程保持空闲等待状态。当主线程遇到并行循环时,循环的不同迭代将会在启动循环执行的从属线程和主线程之间分布。在每个线程完成其块的执行之后,将与剩余线程保持同步。此同步点称为障碍。在所有线程完成其工作并到达障碍之前,主线程不能继续执行程序的剩余部分。从属线程在到达障碍之后进入等待状态,等待分配更多的并行工作,而主线程继续执行该程序。在此期间,可发生多种开销:同步和工作分配的开销,障碍同步的开销。通常存在某些特殊的并行循环,为其执行的有用工作量不足以证明开销是值得的。对于此类循环,其运行速度会明显减慢。

4.1 私有标量和私有数组

对于某些数据依赖性,编译器仍能够并行化循环。例如:

```
for (i=1; i < 10; i++)
  {t = 2 * a[i];b[i] = t;}
```

假定数组 a 和 b 为非重叠数组,而由于变量 t 的存在而使两次迭代之间存在数据依赖性。在第一次迭代和第二次迭代时执行以下语句:

```
t = 2*a[1]; /* 1 */
b[1] = t; /* 2 */
t = 2*a[2]; /* 3 */
b[2] = t; /* 4 */
```

由于第一个语句和第三个语句修改变量 t,因此编译器无法并行执行它们。然而,t 的值始终在同一次迭代中计算并使用,因此编译器可以对每次迭代使用 t 的单独一个副本。这消除了不同迭代之间由

于此类变量而产生的干扰。事实上，我们已使变量 t 成为执行迭代的每个线程的私有变量。这种情形可以说明如下：

```
for (i=1; i < 10; i++)
{
  pt[i] = 2 * a[i]; /* 1 */
  b[i] = pt[i]; /* 2 */
}
```

每个标量变量引用 t 现在被替换为数组引用 pt。现在，每次迭代使用 pt 的不同元素，因此消除了任意两次迭代之间的数据依赖性。本示例产生的一个问题是可能导致数组非常大。在实际运用中，编译器为参与循环执行的每个线程只分配变量的一个副本。事实上，每个此类变量是线程的私有变量。

编译器还可以私有化数组变量，以便为循环的并行执行创造机会。请看以下示例：

```
for (i=1; i < 10; i++)
{
  for (j=1; j < 10; j++)
    {x[j] = 2 * a[i];b[i][j] = x[j];}
}
```

在上例中，外部循环的不同迭代修改数组 x 的相同元素，因此外部循环不能并行化。然而，如果执行外部循环迭代的每个线程均具有整个数组 x 的私有副本，则外部循环的任意两次迭代之间不存在干扰。这种情形说明如下：

```
for (i=1; i < 10; i++)
{
  for (j=1; j < 10; j++)
  {px[i][j] = 2 * a[i];b[i][j] = px[i][j];}
}
```

如私有标量的情形一样，不必为所有迭代展开数组，而只需要达到系统中执行的线程数。这可以由编译器自动完成，完成方式是在每个线程的私有空间中分配初始数组的一个副本。

4.2 返回存储

变量私有化对改进程序中的并行性十分有用。然而，如果在循环外部引用私有变量，则编译器需要确保私有变量具有正确的值。请看以下示例：

```
for (i=1; i < 10; i++)
{
  t = 2 * a[i]; /* 1 */
  b[i] = t; /* 2 */
}
x = t;/* 3 */
```

在上例中，在语句 3 中引用的 t 值是循环计算的最终 t 值。在变量 t 私有化并且循环完成执行之前，t 的正确值需要重新存储到初始变量中。这称为返回存储。这通过将最后一次迭代中的 t 值重新复制到变量 t 的初始位置来完成。在很多情况下，编译器自动执行此操作。但是也存在不易计算最终值的情况。正确执行后，语句 3 中的 t 值通常并不是循环最终迭代中的 t 值。事实上，它是条件为真时的最后一次迭代。通常，计算 t 的最终值十分困难。在类似情况下，编译器不会并行化循环。

4.3 约简变量

有时循环的迭代之间存在真正的依赖性，而导致依赖性的变量并不能简单地私有化。例如，从一次迭代到下一次迭代累计值时，会出现这种情况。在下例中，循环计算两个数组的向量乘积，并将结果赋给一个称为 sum 的公共变量。该循环不能以简单的方式并行化。编译器可以利用语句 1 中的计算的关联特性，并为每个线程分配一个称为 psum[i] 的私有变量。变量 psum[i] 的每个副本均初始化为 0。

```
for (i=1; i < 10; i++)
{sum += a[i]*b[i]; /* 1 */}
```

每个线程以自己的变量 psum [i] 副本计算自己的部分和。在穿过障碍之前，所有部分和均加到初始变量 sum 上。在本示例中，变量 sum 称为约简变量。然而，将标量变量提升为约简变量存在的危险是：累计舍入值的方式会更改 sum 的最终值。只有在专门授权编译器这样做时，编译器才执行该变换。

5 改进循环并行化以提高程序性能

编译器执行多个循环重构变换，帮助改进程序中循环的并行化。其中某些变换还可以提高循环的单处理器执行性能。

5.1 循环分布

循环通常包含少量无法并行执行的语句以及许多可以并行执行的语句。循环分布旨在将顺序语句

移到单独一个循环中，并将可并行化的语句收集到另一个循环中。这一点在以下示例中描述：

```
for (i=0; i < n; i++)
 {
   x[i] = y[i] + z[i]*w[i]; /* 1 */
   a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /* 2 */
   y[i] = z[i] - x[i]; /* 3 */
 }
```

假定数组 x、y、w、a 和 z 不重叠，则语句 1 和 3 可以并行化，但语句 2 不能并行化。可以循环分割或分布为两个不同循环。循环 1 不包含来自初始循环的任何阻止循环并行化的语句，可并行执行。循环 2 包含来自初始循环的不可并行化的语句，不可并行执行。循环分布并非始终有益或安全。编译器会进行分析以确定分布的安全性和有益性。

5.2 循环合并

如果循环的粒度（或循环执行的工作量）很小，则分布的效果可能并不明显。这是因为与循环工作量相比，并行循环启动的开销太大。在这种情况下，编译器使用循环合并将多个循环合并到单个并行循环中，从而增大循环的粒度。当具有相同行程计数的循环彼此相邻时，循环合并很方便且很安全。然而，循环合并并非始终安全。如果循环合并产生了合并之前并不存在的数据依赖性，则合并可能会导致错误执行。编译器执行安全性和有益性分析，以确定是否应执行循环合并。通常，编译器可以合并任意多个循环。以这种方式增大粒度有时可以大大改善循环，使其从并行化中获益。

5.3 循环交换

因为并行化循环嵌套的最外层循环发生的开销很小，所以通常更有益处。然而，由于此类循环可能携带依赖性，并行化最外层循环并非始终安全。在下例中，具有索引变量 i 的循环不能并行化，原因是循环的两次连续迭代之间存在依赖性。这两个循环可以交换，并行循环（j- 循环）变为外部循环：交换后的循环只发生一次并行工作分配开销，而先前发生 n 次开销。编译器会执行安全性和有益性分析，以确定是否执行循环交换。

交换前：

```
for (i=0; i < n; i++)
   {
     for (j=0; j <n; j++)
     {a[j][i+1] = 2.0*a[j][i-1];}
   }
```

交换后：

```
for (j=0; j<n; j++)
   {
     for (i=0; i<n; i++)
     {a[j][i+1] = 2.0*a[j][i-1];}
   },
```

6 使用 Pragma 显式并行化

通常编译器没有足够的信息来判断并行化的合法性或有益性。编译器支持 pragma，使编程人员能够有效地并行化循环，否则编译器很难或根本无法处理这些循环。有两个串行 pragma，均适用于 for 循环：#pragma MP serial_loop，#pragma MP serial_loop_nested。#pragma MP serial_loop pragma 向编译器指示：下一个 for 循环不自动并行化。#pragma MP serial_loop_nested pragma 向编译器指示：下一个 for 循环以及该 for 循环的作用域内嵌套的任何 for 循环均不自动并行化。有一个并行 pragma：#pragma MP taskloop [options]。MP taskloop pragma 可以根据需要带一个或多个以下参数。例如，#pragma MP taskloop maxcpus (number_of_processors) 指定要用于此循环的处理器数。maxcpus 的值必须为正整数。如果 maxcpus 等于 1，则指定的循环将串行执行。使用 #pragma MP taskloop savelast 视为返回存储变量的所有私有变量。#pragma MP taskloop reduction (list_of_reduction_variables) 指定出现在约简列表中的所有变量均将视为循环的 reduction 变量。reduction 变量的部分值可由处理循环迭代的每个处理器单独计算，其最终值可从其所有部分值中计算。reduction 变量列表的存在便于编译器识别循环是否为约简循环，从而允许为其生成并行约简代码。

Sun ISO C 编译器支持多种 pragma 与 taskloop pragma 配合使用，以控制给定循环的循环调度策略。在 static 调度中，循环的所有迭代均匀地分布在参与处理的所有处理器之间。

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(static)
```

```
for (i=0; i<100; i++)
{...}
```

在以上示例中，四个处理器中的每个处理器将处理循环的 25 次迭代。

在 self 调度中，每个参与处理的处理器处理固定次数的迭代(称为“块大小”)，直到循环的所有迭代均已处理完毕为止。

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(self(15))
for (i=0; i<100; i++)
{...}
```

在以上示例中，分配给每个参与处理的处理器的迭代次数按工作请求顺序依次为：

15,15,15,15,15,15,10

在 guided self 引导自我调度中，每个参与处理的处理器处理可变次数的迭代（称为“最小块大小”)，直到循环的所有迭代均已处理完毕为止。

7 建议

如果编译器并行化所花时间量占主体的程序部分会发生加速。例如，如果并行化一个占用程序执行时间的百分之五的循环，则总加速仅限于百分之五。然而，根据工作量和并行执行开销的大小，也可能程序没有任何改善。一般说来，并行化的程序执行所占的比例越大，加速的可能性就越大。每个并行循环均会在启动和关闭期间发生少量开销。启动开销包括工作分配代价，关闭开销包括障碍同步代价。因此，如果大量程序执行工作由许多短并行循环完成，则整个程序可能减慢。编译器可以执行增大循环粒度的循环变换，如循环交换和循环合并。所以尝试进行某些调节，重新构造程序，就可以从并行 C 中获益。 ◇

(上接第 13 页)

时，假定每一个发动机都连续工作，并且喷气时间都与遥测数据采样间隔一致，那么得到的最大等效加速度的理论值为：

$$Acc_{max}=0.01814052(m/s^2)$$

6.3 等效加速度连续状态积分计算

在实际情况中，动量轮卸载均使用离散的遥测采样数据计算，下面讨论等效加速度的解析解。如果发动机是连续工作，那么将式(1.3)求和计算转换为卸载时间段内的连续积分计算，假定一个发动机连续工作，根据式(1.3)计算得：

$$a_{in}=0.00405402(m/s^2)$$

同时将式(1.3)求和计算转换为卸载时间段内的连续积分计算，$\Delta T_d(t)$转化为时间的微分，$\Delta T_d(t)=dt$；$m(t)$是时间的一次函数，令 $m(t)=M-\omega t$，式(1.3)就可转化为：

$$\int_t \vec{a}_{in}(t)=\int_t RM^T\cdot\vec{a}_{bo}(t)=\int_t RM^T\cdot\frac{\vec{f}_{bo}}{m\cdot\Delta T_{scp}}dt \quad (5.1)$$

对(5.1)化简之后得到：

$$a_{in}=\int_t a_{in}(t)=\int_t\frac{F}{m(t)\cdot\Delta T_{scp}}dt \quad (5.2)$$

通过(5.2)式计算可得 $a_{in}=0.00400095(m/s^2)$：

7 结束语

综上所述，就航天器的动量轮卸载过程中的质量误差，判断卸载时间误差，对发动机角度误差进行了分析计算，将计算精度提高到 10^{-4} 量级；并根据遥测数据作了最大等效加速度的评估，还对离散的遥测数据进行了连续性分析，得到了动量轮卸载过程中比较全面的分析数据。如果还需要高精度，通过姿态四元素计算航天器本体系到惯性坐标系的转换矩阵 RM^T 还有待进一步分析。 ◇

参 考 文 献

[1] 刘林. 航天器轨道确定. 国防工业出版社. 2000

[2] 章仁为. 卫星轨道姿态动力学与控制. 北京航空航天大学出版社. 2005

[3] 周军. 航天器控制原理. 西北工业大学出版社. 2001

[4] 无线电遥测遥控(上). 国防工业出版社. 2001